A Computational Method and Software Development for Make-To-Order Pricing Optimization

by

Zhiyong Wang

B. Arch. Tsinghua University, China, 1993
M.S. Texas A&M University, USA, 1998

Submitted to the System Design and Management Program
in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Engineering and Management

at the

Massachusetts Institute of Technology

June 2006

Signature of Author _____
System Design and Management Fellows Program
June 2006

Certified by _____
David Simchi-Levi
Professor of Engineering Systems
Co-Director, LFM/SDM
Thesis Supervisor

Accepted by _____
Patrick Hale
Director, SDM Fellows Program

A Computational Method and Software Development for Make-To-Order Pricing Optimization
by

Zhiyong Wang

Submitted to the System Design and Management Program in
Partial Fulfillment of the Requirements for the Degree of

Master of Science in Engineering and Management

ABSTRACT

High variability of demand and inflexible capacity are inevitable in a make-to-order production despite its cost savings. A computational method is proposed in this thesis to exploit pricing opportunities in the price elasticity of demand and the up-to-date order transactions. Software development possibility was considered based on such pricing optimization method. Based on experiments conducted using a software prototype, we concluded that using the proposed computational method and software developed following the method with acceptable performance and scalability, pricing optimization was able to increase the revenue of a make-to-order production.

Thesis Supervisor: David Simchi-Levi
Title: Professor of Engineering Systems, Co-Director, LFM/SDM

ACKNOWLEDGMENTS

Special thanks to Dr. Simchi-Levi who provided a tremendous amount of support and guidance.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter I – INTRODUCTION

1.1 Background

According to one of the fundamentals of microeconomics, the balance of supply and demand determines the price of a product. In a market economy, when the asking price and the bid price are aligned, a product is traded. In a free market and its long history, pricing has been one of the few strategically important ways for product makers or sellers to attract demand and to out-smart their competitors for market shares. Companies have dedicated personnel in strategic planning, and the marketing department closely monitors how their pricing policies and practices affect the bottom line. They work very hard to set the prices right for their products to compete better in the future.

At the same time, the maturity of streamlined manufacturing processes and information technologies (IT) has enabled many product makers to sell their products in a make-to-order fashion. When delivery lead time is tolerable for the customers and/or sufficient demand maintains the scale of economy of mass production, the make-to-order model works well because it greatly reduces inventory cost and alleviates the impact of demand uncertainty.

1.2 Problem Statement

Despite the efforts in pricing and success in make-to-order operations, there is still an inevitable conflict between ever changing demand and relatively inflexible production capacity. Furthermore, for companies that set fixed product prices or react to demand change slowly with their product prices, there are money and market share to lose when they fail to capture the opportunities from the price elasticity of demand. With very valuable sales transactions and operational data sitting in their IT systems, companies need effective and responsive pricing models and tools to stop profit leaks and gain a competitive advantage in the market [4][5].

1.3 Hypothesis

This thesis suggests that computational method and software products based on such a method can be developed to take advantage of on-going sales transactions and operational data in the IT systems in a business and to exploit the opportunities in price elasticity of demand by mathematically optimizing prices for its make-to-order products, with incremental revenue and profits as the results.

1.4 Research Objectives

This thesis describes the author's three-step objectives: First, to exploit the pricing optimization opportunities for make-to-order products with a proposed computational method; second, to explore the feasibility of designing and implementing a software solution if there are pricing optimization opportunities with the proposed method; and third, to determine whether such a solution could increase revenue and improve profitability for a business if the software solution could be developed.

1.5 Research Methodology

The research methodologies were closely based on the author's three-step objectives. First, to explore the pricing optimization opportunities for make-to-order products, we propose a computational method then use a small but representative example to compare revenue gain between fixed price and optimized price with varying product demand but static product capacity. Second, to explore the feasibility of designing and implementing a software solution, we attempt to develop a software prototype based on the proposed computational method. Third, to determine the pricing benefit from such a computational method and software solution, we conduct a series of analyses with randomized data on the correctness, performance and scalability of the method and software prototype, and more importantly, verify whether the pricing optimization provides statistically significant increase in revenue and profit over fixed pricing.

1.6 Scope and Limitation

The research is limited to the scope of computational pricing optimization and its software development for make-to-order products. The thesis discusses in detail the pricing optimization related topics, such as price elasticity of demand, demand forecast and market segmentation, but there is no attempt to develop computational models in these areas. The thesis also discusses how make-to-order product strategy could affect pricing optimization but there is no attempt to discuss the implementation of make-to-order product strategy per se. Pricing optimization and revenue management (also known as yield management) are related fields in operations research; therefore, some of the revenue management concepts are discussed.

Chapter II – PRICING STRATEGY

2.1 Pricing Strategies [7]

In early 1990s, an article in the McKinsey Quarterly stated a stunning finding: "A price rise of 1% would generate an 11% increase in operating profits; more than 3 times greater than the impact of a 1% increase in volume" [3]. Either to maintain sustainable competitive advantage or to simply survive in competitive market, a business needs to command pricing power without driving customers away and losing market share. There are many ways to strategically plan and determine the price of a product, such as competition based pricing, cost plus pricing and market segment pricing. Competition and cost are commonly used price drivers. Market segment pricing, also known as pricing discrimination, tries to exploit differences of willingness to pay in different market segments. The possibility of exploiting such differences can be explained by price elasticity of demand and market segmentation.

2.2 Price Elasticity of Demand [6]

One of the established theories of economics, or simply common sense in the free market, tells us that demand reacts to price. Low price attracts demand while high price drives demand away. The price-demand curve is downward sloping, as Figure 1 illustrates. Pricing elasticity of demand (PED) is how sensitive the change of demand responds to the change of price. The PED is calculated as the absolute percentage change in price divided by the percentage change in demand.

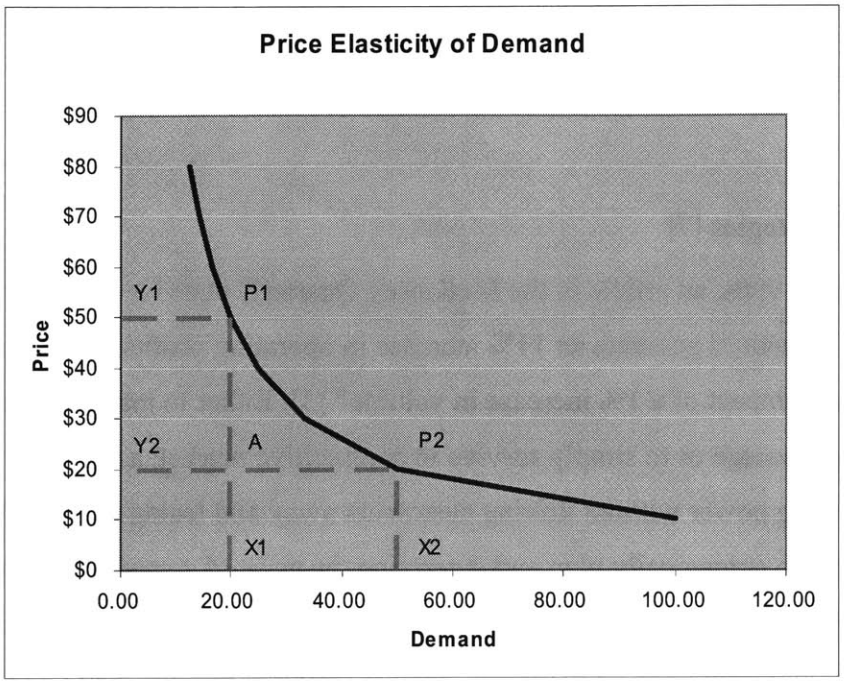

Figure 1. Price elasticity of demand

When the price elasticity is high (PED > 1), demand drops at a faster rate than price rises; thus, total revenue decreases. When the price elasticity is low (PED < 1), demand drops at a slower rate than price rises; thus total revenue increases. In Figure 1, P1 and P2 represent low and high price elasticity cases, respectively.

2.3 Market Segmentation

If price elasticity of demand can result either in total revenue increase or total revenue decreases, how can a pricing strategy exploit price elasticity to gain more profit? The answer to this question lies in the market segmentation. Figure 2 illustrates the same price-demand curve as in the previous section, but both price P1 and price P2 are offered to the market targeting two separated market segments, respectively.

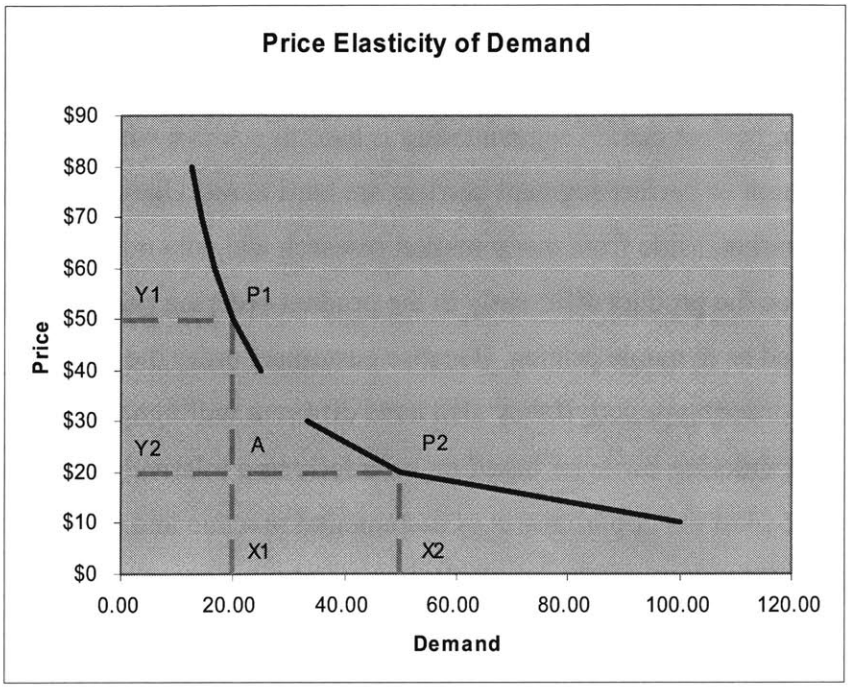

Figure 2. Price elasticity of demand with market segmentation

When only one price, either P1 or P2, is offered to the market, the revenue is either the area of (O,Y1,P1,X1) or the area of (O,Y2,P2,X2). When both prices are offered to two separate market segments, there is incremental revenue of either the area of (X1,A,P2,X2) adding to the area of (O,Y1,P1,X1), or the area of (Y2,Y1,P1,A) adding to the area of (O,Y2,P2,X2). Hypothetically, if the markets were to be infinitely segmented, the maximal revenue that could have been achieved with the price-demand curve in Figure 2 is the entire area under the curve itself. Successful market segmentation requires homogeneity within each segment, heterogeneity among segments, measurability, accessibility and large enough scale to be profitable in each segment. In most cases, there have to be significant or arbitrary barriers to segment the market. The examples are regular retail price versus academic retail pricing, and Saturday-night-stay rule to separate leisure travelers from business travelers.

2.4 Dynamic Pricing [8]

To exploit the pricing opportunities, price elasticity and market segmentation are needed. However, perfect market segmentation is hard to achieve when information becomes transparent or perfect segment barriers are hard to set. One of the natural ways to segment the market, aside from many market research and data mining efforts, is to use time, i.e., price the product differently in the product ordering period. Such a pricing approach is termed as dynamic pricing. Because customers order the product at different times within the ordering period, if they also have different willingness to pay for the product and their ordering behavior based on time follows a relatively predictable pattern, it is possible to exploit the opportunities of incremental revenue and profit by using dynamic pricing.

2.5 Pricing in Make-To-Order Production

Successful make-to-order production relies on attraction of product customization and tolerance towards the delivery lead time, yet maintains mass production for scale of economy and flexible production line for customization even though the total capacity of the product is not flexible. Product customization provides market segments such as the premium product segment and the economical product segment, or the residential product segment and the commercial product segment. Different levels of tolerance towards the delivery lead time also segment the market with different ordering pattern based on order time. Assuming price elasticity of demand for make-to-order products exists, it is possible to gain incremental revenue and profit through dynamic pricing in the product ordering period.

Chapter III – PRICING OPTIMIZATION AND ITS COMPUTATIONAL METHOD

3.1 Demand Forecast

Even with a make-to-order production model, there is an inevitable conflict between ever changing demand and relatively inflexible production capacity. Product capacity is usually planned for long-term, and large scale capacity expansion is usually costly and time-consuming. In some cases, companies are reluctant to expand their product capacity because over-capacity production may incur higher operational risk or higher potential of financial loss than under-capacity production. Demand forecast is critical when such conflict is eminent.

Demand forecast will never exactly match the actual demand but its accuracy can be improved. Forecasting demand at aggregated level and/or forecasting short-term demand based on up-to-date actual data can possibly improve the accuracy of the prediction. A make-to-order product model requires a relatively short-term demand forecast because of relative short delivery lead time, and more importantly, it provides up-to-date ordering information that can greatly improve the accuracy of demand forecasts when the forecast model is responsive enough.

One of the modern statistical models that can take the advantage of the up-to-date information is the Bayesian statistical model. Based on Bayes' theorem, a prior probability $\mathbf{P(H_i)}$, of one of mutually exclusive and exhaustive events, can be updated when an evident **E** becomes available. The posterior probability of such an event is the product of prior probability and likelihood of the event [1].

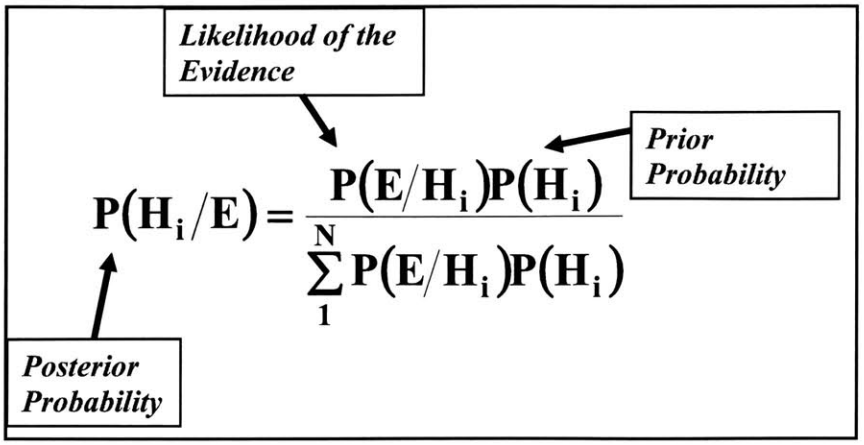

Figure 3. Bayes' theorem

Here is an example of how Bayes' theorem applies to make-to-order production. Assume the order rate λ for a product per certain ordering period **t** has the following probability mass function: {8, 0.2}, {9, 0.5}, and {10, 0.3}. The order volume is assumed to be a Poisson distribution:

$$P[k, t \mid \lambda] = e^{-\lambda t} (\lambda t)^k / k!$$

When a quarter of this ordering period has passed, there are six orders on hand. The following table shows how prior probability is updated based on the available evidence (where $k = 6$ and $t = ¼$). Apparently, such evidence increases the probability of one order rate while decreased probability of the other two order rates. In other words, the probability of having 10 as the final demand increased based on up-to-date order information.

Order Rate (λ)	Prior Probability ($P_0(\lambda)$)	Likelihood of Evidence ($P(E\|\lambda)$)	$P_0(\lambda) * P(\lambda\|E)$	Posterior Probability $P_1(\lambda\|E)$
8	0.200	0.012	0.0024	0.118
9	0.500	0.019	0.0095	0.468
10	0.300	0.028	0.0084	0.414
Total	1.000		0.0203	1.000

Table 1. An example of updating probability using Bayes' theorem

There are two important aspects in the example above using Bayesian statistical model. One aspect is that up-to-date information can update the probability of the previous belief. This is particularly suitable in make-to-order production because when orders come in, the demand forecast can be updated accordingly until the production starts. The other aspect is that a Bayesian model is computationally feasible to use only the prior probability and the probability mass (or distribution) function to calculate posterior probability. However, outlier detection is critical when using a Bayesian model because outliers may steer the demand forecast model in the wrong direction.

3.2 Pricing Optimization

When we have production capacity planned and demand forecasted, we can easily construct a linear programming (LP) model on how much demand we can take for each product while maximizing the total revenue. Such models are well researched in the field of revenue management. After all, the goal of the revenue management is to sell the right product (or service) at the right time using the right resource when the resource is perishable. The capacity in a make-to-order production is perishable because idle capacity means there are not enough orders before the production starts, hence opportunity costs or financial losses incur. On the other hand, filling the capacity with low revenue contribution orders is undesirable because they dilute the value of the limited resources [2].

An LP model borrowed from the revenue management field can be used as the first step towards pricing optimization. Here are the notations and formulation for the LP model:

- **P**: Set of products
- **A**: Set of prices for each product can be sold at
- **R**: Set of resources to produce P
- **T**: Set of future time units
- $\mathbf{X_{p,a,r,t}}$: Decision variable for product p selling at price a to be produced by resource r and to be delivered at time t

- $\alpha_{p,a,r,t}$: Demand value for product *p* selling at price *a* to be produced by resource *r* and to be delivered at time *t*
- $a_{p,a,r,t}$: Resource unit usage for product *p* selling at price *a* to be produced by resource *r* and to be delivered at time *t*
- $C_{r,t}$: Capacity for resource *r* at time *t*
- $D_{p,a,t}$: Demand for product *p* at price *a* delivered at time *t*
- Objective function:
  - Maximize: $\Sigma_P \Sigma_A \Sigma_R \Sigma_T [\alpha_{p,a,r,t} X_{p,a,r,t}]$
- Constraints:
  - $\Sigma_P \Sigma_A [a_{p,a,r,t} X_{p,a,r,t}] \leq C_{r,t'}$ where $t - d \leq t' < t$ and d is production duration
  - $0 \leq \Sigma_R X_{p,a,r,t} \leq D_{p,a,t}$

The LP model assumes that a product can be produced by one of the many resources with multiple time units and multiple resource units. After the LP is solved, the solutions would be the available amounts of each product selling at certain price to be produced by certain resource and to be delivered at certain future time.

When each product can be sold at different prices in perfectly segmented markets, we need to control the availability, not the prices, because we match the price to the willingness to pay in each market segment and make sure we only take the orders up to the availability. It is easy to see demand with high revenue contribution is allocated with enough availability when capacity and demand permits. Demand with low revenue contribution has to use the amount of capacity left even though the customers are willing to tolerate long delivery lead time.

We can easily see the flaws of this model. The first and foremost is that perfect market segmentation is very hard to achieve and may not exist in real life or may have legal consequences if segment barriers violate the law. Subsequently, it may not be possible to set different prices at the same time point for the same product. However, market segmentation still can be done by product customization and by time. The LP model can be modified to solve dynamic pricing optimization.

The critical issue in dynamic pricing, or setting different prices at different time to maximize the revenue, is that customers with higher willingness to pay can pay a lower price when the price is available to the market. Such a "buy-down" phenomenon changes pricing optimization on many aspects. The first is that demand forecasts for the same product at different prices are no longer independent and can only be updated based on orders that come in at the current price. The second is that in the LP model, decision variables for the same product at different prices cannot be constrained independently by demand forecasts at different price.

To further illustrate the impact of the "buy-down" phenomenon, we use the following example with one product and three days of the ordering period and customer A, B and C with $350, $500 and $800 willingness to pay, respectively. When the capacity is constrained, here are the scenarios that use dynamic pricing to maximize the revenue:

	Demand Pattern Forecasted			Optimal Price			Revenue			
Scenario	Day1	Day2	Day3	Day1	Day2	Day3	Day1	Day2	Day3	Total
1	A	B	C	$ 350	$ 500	$ 800	$ 350	$ 500	$ 800	$ 1,650
2	A	C	B	$ 350	$ 500	$ 500	$ 350	$ 500	$ 500	$ 1,350
3	B	A	C	$ 350	$ 800	$ 800	$ 350		$ 800	$ 1,150
4	B	C	A	$ 350	$ 800	$ 800	$ 350	$ 800		$ 1,150
5	C	A	B	$ 350	$ 350	$ 500	$ 350	$ 350	$ 500	$ 1,200
6	C	B	A	$ 350	$ 350	$ 350	$ 350	$ 350	$ 350	$ 1,050
7	AB	C		$ 350	$ 800	---	$ 700	$ 800		$ 1,500
8	AB		C	$ 350	$ 800	$ 800	$ 700		$ 800	$ 1,500
9	AC	B		$ 350	$ 500	---	$ 700	$ 500		$ 1,200
10	AC		B	$ 350	$ 500	$ 500	$ 700		$ 500	$ 1,200
11	BC	A		$ 350	$ 350	---	$ 700	$ 350		$ 1,050
12	BC		A	$ 350	$ 350	$ 350	$ 700		$ 350	$ 1,050
13	A	BC		$ 350	$ 500	---	$ 350	$ 1,000		$ 1,350
14	A		BC	$ 350	$ 500	$ 500	$ 350		$ 1,000	$ 1,350
15	B	AC		$ 350	$ 800	---	$ 350	$ 800		$ 1,150
16	B		AC	$ 350	$ 800	$ 800	$ 350		$ 800	$ 1,150
17	C	AB		$ 350	$ 350	---	$ 350	$ 700		$ 1,050
18	C		AB	$ 350	$ 350	$ 350	$ 350		$ 700	$ 1,050
19		A	BC	$ 350	$ 350	$ 500		$ 350	$ 1,000	$ 1,350
20		B	AC	$ 350	$ 350	$ 800		$ 350	$ 800	$ 1,150

21		C	AB	$ 350	$ 350	$ 350		$ 350	$ 700	$ 1,050
22		AB	C	$ 350	$ 350	$ 800		$ 700	$ 800	$ 1,500
23		AC	B	$ 350	$ 350	$ 500		$ 700	$ 500	$ 1,200
24		BC	A	$ 350	$ 350	$ 350		$ 700	$ 350	$ 1,050
25	ABC			$ 350	---	---	$ 1,050			$ 1,050
26		ABC		$ 350	$ 350	---		$ 1,050		$ 1,050
27			ABC	$ 350	$ 350	$ 350			$ 1,050	$ 1,050

Table 2. An example of maximizing revenue with dynamic pricing

In this example, the maximum revenue with perfect market segmentation is $1650. With the "buy-down" phenomenon, the average of maximized revenue in all 27 scenarios is about $1207. Although it is lower than $1650, it is higher than $1050 as the revenue with the best fixed price can be offered. The formulation of LP to solve the dynamic pricing optimization problem is as follows with the notations specified previously:

- Objective function:
  - Maximize: $\Sigma_P \Sigma_A \Sigma_R \Sigma_T [\alpha_{p,a,r,t} X_{p,a,r,t}]$
- Constraints:
  - $\Sigma_P \Sigma_A [a_{p,a,r,t} X_{p,a,r,t}] \leq C_{r,t'}$ where $t - d \leq t' < t$ and **d** is production duration for each **p** and **t**
  - $0 \leq \Sigma_R \Sigma_A [(1/ D_{p,a,t}) X_{p,a,r,t}] \leq 1$ for each **p** and **t**

The second constraint reflects that even though a product to be delivered at certain time can be sold at different price, only one price is offered to the market at a single time point of optimization. The demand used in this constraint is the demand with the willingness to pay equal or higher than the price point. The solutions of this LP model are still availability for each product at each price, but since we can only offer one price for one product at a given ordering time, the price cannot be simply deduced from the solutions of the decision variables because it is possible to have non-zero solutions for decision variables of the same product at different prices.

3.3 Bid Price Control

To set the optimal price at a certain time in the ordering period for each product to be delivered at certain time, the marginal values, or duals, from the LP model in the previous section need to be used [2]. Here are the notation and formulation to generate optimal prices:

- $\mathbf{Y_i}$: Marginal values for each constraint
- $\mathbf{BP_{p,a,t}}$: Bid price for each product *p* at price *a* to be delivered at time *t*
- $\mathbf{BP_{p,t}}$: Bid price for each product *p* to be delivered at time *t*
- Bid price calculation:
  - $\mathbf{BP_{p,a,t}} = \mathbf{\Sigma_R} [\mathbf{a_{p,a,r,t}} \mathbf{Y_i}] + (\mathbf{Y_i} / \mathbf{D_{p,a,t}})$ and
  - $\mathbf{BP_{p,t}} = \mathbf{A}$ where A is the lowest price that has $\mathbf{A} \geq \mathbf{BP_{p,a,t}}$

Delivery Lead Time	Product A	Product B	Product C	Product D
X	Sold Out	$ 49.99	$ 79.99	$ 129.99
Y	$ 29.99	$ 49.99	$ 69.99	$ 119.99
Z	$ 29.99	$ 39.99	$ 69.99	$ 109.99

Table 3. An example of bid price table for make-to-order products

Assuming the current time is $\mathbf{t_0}$, $\mathbf{t} - \mathbf{t_0}$ can be interpreted as delivery lead time. After the LP problem is solved and bid price is generated, the make-to-order production will have a bid price table similar to the example in Table 3. The bid price table will be updated after the demand forecasts are updated and the pricing optimization is executed.

Chapter IV – SOFTWARE DESIGN AND IMPLEMENTATION

4.1 Software Requirements and Architecture

To develop a software system based on the computational method described in Chapter III, we need the system requirements, scope of the requirements and software architecture design before we can implement it. The requirements for the software system development are summarized as follow:

- The system shall use order transactions as its input from an IT system that controls sales order in a make-to-order production
- The system shall forecast demand based on the order transaction
- The system shall update forecast demand as frequently as it inputs order transactions
- The system shall use production capacity plans as its input
- The system shall optimize prices for make-to-order products once demand forecasts or capacity plans are updated
- The system shall provide optimal bid price controls as its output to the IT system that controls sales orders

To emphasize the pricing optimization, the scope of requirements is limited to the last two items as was stated in Chapter I.

The software architecture consists of three parts. The backend relational database management system (DBMS) manages the inputs, outputs, intermediate data if any, and data transaction and recovery processes; the optimization engine, interfaced with the DBMS, builds the data sets and their relations and executes the pricing optimization logic; the general LP solver connects the optimization engine and the existing commercial off-the-shelf or open source LP solver.

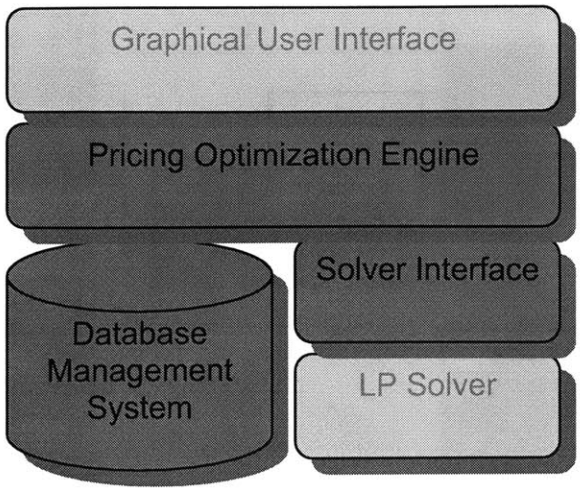

Figure 4. System architecture with external components

4.2 Database Design

Considering the system as whole, we would need database tables to hold the following data sets including transaction inputs, pricing control outputs and demand forecast and inventory plans as intermediate data:

- Order transactions
- Demand forecast
    - Detail forecast for future orders
    - Seasonality and special event adjustments
- Demand forecast and order on-hand summaries
- Product definitions
- Product-resource mappings
- Resource capacity plans
- Product bid price controls

To focus on pricing optimization, the database schema of the last five tables is illustrated in detail in Figure 5.

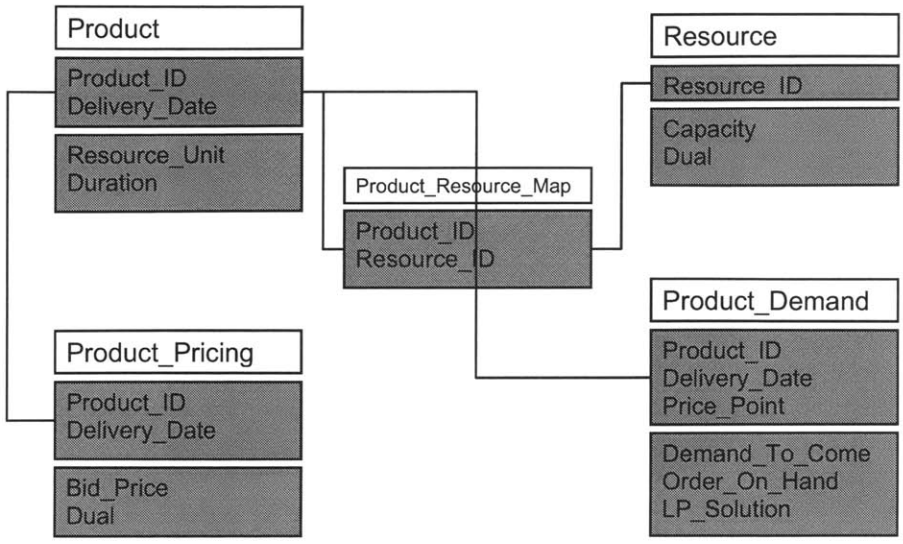

Figure 5. Database tables and their relationships

4.3 Data Type and Algorithm

Two critical data types for the optimization engine are designed for performance and scalability reasons One is a general purpose "data container"; the other is a general purpose "relation" (see Appendix A). The "data container" contains a hash table to facilitate quick look-up to any single "row" of data based on the hash key, while the "relation" contains hash maps to facilitate quick one directional look-up from one data container to another based their one-to-many relationship. All special purpose data sets are inherited from the "data container" and are designed as column-major instead of row-major data collection so that they all can grow row-wise while stay with pre-defined columns. Such data type design is fairly scalable when the size of the optimization problem increases or decreases because the number of internal objects stays as the fixed number of columns while row-wise growth is merely memory allocation without additional object creation.

Since one LP problem by nature cannot be divided into sub-problems, the data sets for the LP formulation in pricing optimization are not easy to scale. One challenge in a large scale LP problem is the potentially large memory footprint for the matrix that maps decision variables to constraints. The general LP solver interface assumes that any

particular external LP solver is able to handle sparse matrix data type in a profile like implementation (see Appendix A). Such implementation contains only non-zero values in a one-dimensional array with column-major count and row-position index. The "relation" data type matches this implementation fairly well, so that there would be no specific algorithm but simple hash look-up and quick random access of arrays to populate the sparse matrix using the "relation" data type.

4.4 Implementation

The first step of the implementation of the pricing optimization system is to build a prototype for the pricing optimization engine. The prototype assumes the data sets of demand forecasts and resource capacity plans are ready to use. The major "data containers" implemented in the prototype closely follow the computational method proposed in Chapter III. They are "product" as **P**, **T** and **A** set, "product pricing" as $\mathbf{BP}_{p,t}$ set, "product decision" as the combination of $\boldsymbol{\alpha}_{p,a,r,t}$, $\mathbf{a}_{p,a,r,t}$, and $\mathbf{X}_{p,a,r,t}$, and "resource" as the combination of **R** set and $\mathbf{C}_{r,t}$ set. "Pricing optimizer" is the optimization engine that builds the "relations" among the data sets and executes the pricing optimization logic (see Appendix B).

Chapter V – EXPERIMENTS AND ANALYSIS

5.1 Design of Experiment

Based on the computational method described in Chapter III and software design and implementation methodology described in Chapter IV, we developed a software prototype to facilitate the experiments and analysis. There are three groups of experiments designed to verify the hypothesis of this thesis:

- Experiments to verify the correctness of the computational method and the software prototype
- Experiments to verify the performance and scalability of the software prototype
- Experiments to verify whether the pricing optimization using the proposed computational method and the software prototype can result in incremental revenue compared to fixed pricing.

The software prototype was implemented by using java programming language with Java^{tm} 2 Standard Edition 5.0. The hardware used for the experiments was a personal computer with Intel Pentium-4® processor at 2.5 GHz and 1 GB Random Access Memory (RAM) running Microsoft Windows 2000© Professional operating system.

5.2 Subjective Analysis

The first group of experiments is to verify the correctness of the proposed computational method and the software prototype. A notation such as "Price A ($350, 15)" indicates a product that is offered at Price A, which is $350, with demand of 15 orders. Note that even though a product can be offered at multiple prices, only one price should be available at any given time, and demand at any price point is the demand that has willingness to pay equal or higher than the price; in other words, demand at a lower price point always includes demand at a higher price point, so that the maximum demand

a product can attract is the demand at the lowest price point, not the sum of demand at all price points.

The first two experiments show how capacity affects the price in pricing optimization when price elasticity of demand (PED) is greater than or less than 1, respectively. The results show that the optimal prices match expected prices with both capacity and demand constraints affecting pricing decisions. When PED is greater than 1, the optimal price stays low until there is not enough capacity. When PED is less than 1, the optimal price stays high to prevent the "buy down" from diluting the value of the limited production capacity.

	LP Solution					
Capacity	Price A ($350, 15)	Price B ($500, 10)	Price C ($800, 5)	Bid Price	Optimal Price	Expected Price
16	15	0	0	$ 333.33	$ 350.00	$ 350.00
15	15	0	0	$ 333.33	$ 350.00	$ 350.00
14	12	2	0	$ 350.00	$ 350.00	$ 350.00
13	9	4	0	$ 350.00	$ 350.00	$ 350.00
12	6	6	0	$ 350.00	$ 350.00	$ 350.00
11	3	8	0	$ 350.00	$ 350.00	$ 350.00
10	0	10	0	$ 500.00	$ 500.00	$ 500.00
9	0	8	1	$ 500.00	$ 500.00	$ 500.00
8	0	6	2	$ 500.00	$ 500.00	$ 500.00
7	0	4	3	$ 500.00	$ 500.00	$ 500.00
6	0	2	4	$ 500.00	$ 500.00	$ 500.00
5	0	0	5	$ 800.00	$ 800.00	$ 800.00
4	0	0	4	$ 800.00	$ 800.00	$ 800.00
3	0	0	3	$ 800.00	$ 800.00	$ 800.00
2	0	0	2	$ 800.00	$ 800.00	$ 800.00
1	0	0	1	$ 800.00	$ 800.00	$ 800.00
0	0	0	0	Sold Out	N/A	N/A

Table 4. Comparing optimization and expected results on capacity effect (PED > 1)

	LP Solution					
Capacity	Price A ($350, 15)	Price B ($500, 11)	Price C ($800, 7)	Bid Price	Optimal Price	Expected Price
16	0	0	7	$ 785.71	$ 800.00	$ 800.00
15	0	0	7	$ 785.71	$ 800.00	$ 800.00
14	0	0	7	$ 785.71	$ 800.00	$ 800.00
13	0	0	7	$ 785.71	$ 800.00	$ 800.00
12	0	0	7	$ 785.71	$ 800.00	$ 800.00
11	0	0	7	$ 785.71	$ 800.00	$ 800.00
10	0	0	7	$ 785.71	$ 800.00	$ 800.00
9	0	0	7	$ 785.71	$ 800.00	$ 800.00
8	0	0	7	$ 785.71	$ 800.00	$ 800.00
7	0	0	7	$ 800.00	$ 800.00	$ 800.00
6	0	0	6	$ 800.00	$ 800.00	$ 800.00
5	0	0	5	$ 800.00	$ 800.00	$ 800.00
4	0	0	4	$ 800.00	$ 800.00	$ 800.00
3	0	0	3	$ 800.00	$ 800.00	$ 800.00
2	0	0	2	$ 800.00	$ 800.00	$ 800.00
1	0	0	1	$ 800.00	$ 800.00	$ 800.00
0	0	0	0	Sold Out	N/A	N/A

Table 5. Comparing optimization and expected results on capacity effect (PED < 1)

The second group of tests focused on the performance and scalability of the software prototype. The main concern for software development in pricing optimization, or in mathematical solution applications in general, is that when the size of the problem grows, the execution time and memory usage could grow faster based on its underlying algorithm or computational method. If that is the case, we cannot expect software solutions to perform well or scale well in real life situations. For the computational method proposed in this thesis, Table 6 shows how quickly the size of the pricing problem can grow when just one related quantity grows. The size related quantities include number of products, number of price points, number of days in optimization horizon, number of resources (production lines) products can share and duration to produce each type of the products.

Tests	Number of Products	Number of Price Point	Number of Future Delivery Dates	Number of Resources per Product	Average Production Duration	Total Number of Decision Variables	Total Number of Constraints	Total Number of Non-Zeros in the Matrix
1	10	5	10	4	2	2000	140	5800
2	10	5	100	4	2	20000	1400	59800
3	100	5	100	4	2	200000	10400	598000
4	10	5	10	4	8	2000	140	12400
5	10	5	100	4	8	20000	1400	174400
6	100	5	100	4	8	200000	10400	1744000

Table 6. Size of the software performance and scalability experiments

However, there is one important factor in the scalability of pricing optimization. When two resources, or production lines, are not shared by any products, they can be put in the separate optimization processes. Hence, the total number of resources may not relate to the size of the problem because if each of them produces mutually exclusive products, each of the resource with the products it produces is an independent pricing optimization process. Table 7 shows execution time and memory usage of six tests. The first and second group of three tests shows ten times of size increases from on to the other. Neither execution time nor memory usage grows in the same scale. It demonstrates reasonable performance and good scalability for the software prototype based on the method this thesis proposed. The test pairs of 1 and 4, 2 and 5, and 3 and 6 are similar size of the problem but different density of the matrix, the relations between decision variables and constraints. They scale very well except for execution time comparing test 2 and 5. This indicates that the density of the matrix, or the degree of interaction between decision variables and constraints, has significantly larger impact on the scalability of execution time than the number of decision variables and constraints.

Tests	Total Number of Decision Variables	Total Number of Constraints	Total Number of Non-Zeros in the Matrix	Density of the Matrix	Execution Time (second)	Memory Usage (MB)
1	2000	140	5800	0.0207	1.4	70
2	20000	1400	59800	0.0021	2.4	80
3	200000	10400	598000	0.0003	11.0	150
4	2000	140	12400	0.0443	1.6	75
5	20000	1400	174400	0.0062	11.8	85
6	200000	10400	1744000	0.0008	16.0	220

Table 7. Results of the software performance and scalability experiments

The third group of experiments focused on how pricing optimization affects the revenue. This group of experiments had pricing optimization executed for each of the 10 days in the ordering period prior to production. After the optimal price was set day by day, a certain number of orders were accepted with "buy down" if there was any. The remaining capacity and changing demand pattern affect the optimal price for each subsequent day.

The first few experiments demonstrate how predictable demand patterns can significantly increase the revenue. When PED is greater than 1, the optimal price starts low. If the demand that reacts to the low price comes early in the ordering period, "buy down" happens at a lesser degree. PED subsequently decreases for the remaining demand to come and the optimal price rises to the next point after enough low price demand taken. When PED is less than 1, the optimal price starts high. If the demand that is not turned away by high price comes early, the orders that pay premium are protected by such high price until the PED increases to the level that lower price can bring more revenue. Table 8 and Table 9 show how these two preferable order patterns affect the optimal prices, respectively. The revenue increase based on the pricing optimization is at a significant 10% level compared to fixed prices set for the entire ordering period.

Total Demand-To-Come by Days Prior to Production				Pricing Optimization	
Days Prior to Production	Price A ($100)	Price B ($120)	Price C ($140)	Optimal Price	Order Accepted
10	100	80	60	$ 100	15
9	85	72	56	$ 100	15
8	70	64	51	$ 100	14
7	56	56	46	$ 120	8
6	48	48	40	$ 120	8
5	40	40	34	$ 120	8
4	32	32	28	$ 140	7
3	24	24	22	$ 140	7
2	16	16	15	$ 140	7
1	8	8	8	$ 140	8
					Increase
Revenue	$ 10,000	$ 9,600	$ 8,400	$ 11,340	13.4%

Table 8. Pricing optimization increases revenue on predictable demand patterns (PED > 1)

Total Demand-To-Come by Days Prior to Production				Pricing Optimization	
Days Prior to Production	Price A ($100)	Price B ($120)	Price C ($140)	Optimal Price	Order Accepted
10	100	90	80	$ 140	9
9	91	81	71	$ 140	9
8	82	72	62	$ 140	9
7	73	63	50	$ 120	9
6	63	54	40	$ 120	9
5	53	45	30	$ 120	9
4	43	36	20	$ 120	9
3	33	27	10	$ 100	10
2	23	18	0	$ 100	11
1	12	9	0	$ 100	12
					Increase
Revenue	$ 10,000	$ 10,800	$ 11,200	$ 11,400	14.0%

Table 9. Pricing optimization increases revenue on predictable demand patterns (PED < 1)

Table 8 implies situations where certain customers paying premium prices demand short delivery lead time, while other customers seeking discounted price tolerate long delivery lead time. On the other hand, Table 9 implies situations where certain customers order desirable (or trendy) products or product configurations with premium

price early, while other customers wait for discounted price with low grade (or less trendy) products or product configurations.

However, demand may not be as predictable as it should be. The next few experiments were based on randomly generated demand with a daily average and roughly 20% level of standard deviation. The total demand level was used to determine the PED. Table 10 shows one example of randomly generated demand for a 10-day order period prior to production and five price points with PED greater than 1.

Total Demand-To-Come by Days Prior to Production					
Days Prior to Production	**Price A ($100)**	**Price B ($120)**	**Price C ($140)**	**Price D ($170)**	**Price E ($200)**
10	1538	1060	900	731	595
9	1398	945	833	655	524
8	1219	850	739	589	478
7	1055	757	669	518	410
6	905	676	568	432	338
5	739	554	493	351	292
4	574	430	391	285	239
3	406	318	308	220	182
2	254	233	212	135	114
1	138	125	99	82	70

Table 10. Randomly generated demand-to-come on 10 days prior with PED > 1

Each of the randomly generated demand sets was optimized against both sufficient and constrained resource capacity on each day in the ordering period in sequence. The number of accepted orders was based on both the optimal price and the corresponding demand. Capacities were adjusted accordingly based on the number of orders accepted. One example of the series of pricing optimization is shown in Table 11 below. The capacity constraint had a significant impact on how the optimal price was determined.

Days Prior to Production	Optimal Price	Order Accepted	Remaining Capacity	Optimal Price	Order Accepted	Remaining Capacity
10	$ 100.00	140	1600	$ 100.00	140	1000
9	$ 100.00	179	1460	$ 100.00	179	860
8	$ 100.00	164	1281	$ 100.00	164	681
7	$ 100.00	150	1117	$ 170.00	86	517
6	$ 100.00	166	967	$ 170.00	81	431
5	$ 100.00	165	801	$ 140.00	102	350
4	$ 100.00	168	636	$ 140.00	83	248
3	$ 100.00	152	468	$ 200.00	68	165
2	$ 140.00	113	316	$ 200.00	44	97
1	$ 120.00	125	203	$ 200.00	53	53

Table 11. Pricing optimization based on the same demand and different capacity

Table 12 shows four experiments with significant revenue increases. The revenue increases based on pricing optimization with randomly generated demand were limited to 8% or less. In a few cases where there was not any PED swing, the optimal price stayed unchanged; thus, there was no revenue increase.

Tests	Price A ($100)	Price B ($120)	Price C ($140)	Price D ($170)	Price E ($200)	Max Revenue at Fixed Price	Optimal Revenue	Percent Increases
1	$ 153,800	$ 127,200	$ 126,000	$ 124,270	$ 119,000	$ 157,800	$ 159,200	**3.52%**
2	$ 100,000	$ 120,000	$ 126,000	$ 124,270	$ 119,000	$ 126,000	$ 135,590	**7.61%**
3	$ 135,400	$ 154,920	$ 171,920	$ 180,540	$ 179,600	$ 180,540	$ 191,500	**6.07%**
4	$ 100,000	$ 120,000	$ 140,000	$ 170,000	$ 179,600	$ 179,600	$ 191,500	**6.63%**

Table 12. Experiments based on randomly generated demand, different capacity and PED

Even though time is a natural way to segment the market, it is still far from perfect market segmentation. The most important factor is the existence of "buy down" regardless of whether we optimize the publicly listed prices or negotiated deal prices because the information of the price becomes more and more transparent. We do not know what the exact willingness to pay of a particular customer is when an order is placed, or a deal is negotiated, unless the price is set at the highest possible point. It is imperative to optimize prices based on demand forecast that contains ordering behavior in days prior to production. "Days prior" forecast provides both the basis of prior

probability and the likelihood of an order evidence; thus, the posterior probability of demand at all price points can be updated based on orders accepted at a given price. More importantly, the reason why dynamic pricing optimization based on time-based segmentation is particularly suitable for make-to-order productions is that the orders accepted for the future product provide just-in-time updates to reveal the potential PED changes. As we learned from the experiments, PED change at days prior to production based on capacity constraints or order updates is a necessary condition to exploit the incremental revenues and profits. Software solutions provide computational logic and power to react to the PED changes promptly in the ordering period in a make-to-order production.

Chapter VI – CONCLUSION

6.1 Conclusion

Based on the experiments conducted using a developed software prototype, it is evident that the computational method this thesis proposes can correctly optimize the prices to generate incremental revenue and profit with high variability of demand and inflexibility of production capacity. Software solutions can be developed based on this method with reasonable performance and good scalability in terms of execution time and memory usage. This computational method and software development is particularly suitable for make-to-order production because it is imperative that ordering pattern changes be promptly grasped and that optimal prices be set for the right products with the right delivery lead times.

6.2 Further Discussion

There are a few pricing optimization related topics calling for further research. The first and foremost one is how to gain insights on pricing elasticity of demand. In the thesis, the price points and their elasticity are assumed to be determined. In reality, with new products coming to the market everyday, the comprehensive knowledge of price elasticity for a particular product is not easy to gain without extensive market and price tests. However, it is possible to take the advantage of a pricing optimization software system to expand the limited product price offerings to test and collect valuable information on how the customers and competition react to them. Another interesting topic is research on the correlation of the demand at adjacent price points. This is not only related to price elasticity but also provides demand forecast models that may be more accurate and responsive in make-to-order productions with the up-to-date order transactions that do not necessarily have exact willingness to pay information.

REFERENCES

1. Berry, D. A. *Statistics: A Bayesian Perspective*. Belmont, CA: Duxbury Press, 1996
2. Hillier, F. S., and G. J. Lieberman. *Introduction to Operations Research. 7th Ed.* New York: McGraw-Hill, 2001
3. Marn, M. V., and R. L. Rosiello. "Managing Price, Gaining Profit." *McKinsey Quarterly*, No. 4, 1992
4. Kay, E. "Optimizing Pricing to Maximize Profits: Price Management Is Costly, But Well Worth the Investment." *Frontline Solutions*, Sept, 2003
5. McPartlin, J. "The Price You Pay: Companies Say Price-Optimization Software Is Worth the Effort It Demands." *CFO*, Spring, 2004
6. "Price Elasticity of Demand." *http://en.wikipedia.org/wiki/Price_elasticity_of_demand*
7. "Price Strategies." *http://en.wikipedia.org/wiki/Price_Strategies*
8. Simchi-Levi, D.; P. Kaminsky; and E. Simchi-Levi. *Designing & Managing the Supply Chain: Concepts, Strategies, and Case Studies, 2nd Ed.* New York: McGraw-Hill, 2003

APPENDIX A

A.1 General Purpose Data Types for Pricing Optimization

A.1.1 Data Container

```
/**
 * @author wangz@mit.edu
 * @version 1.0
 */
import java.util.Hashtable;
import cern.colt.list.IntArrayList;

public class DataContainer {
  // General-Purpose Data Container
  protected int _iSize;
  protected Hashtable _hash;

  public DataContainer(int iSize) {
    _iSize = iSize;
  }

  public int getSize() {
    return _iSize;
  }

  public IntArrayList getRowsByKey(String sKey) {
    if (_hash == null)
      return null;
    return (IntArrayList)_hash.get(sKey);
  }

  public int getHashSize() {
    return (_hash == null)?0:_hash.size();
  }

  public Hashtable getHash() {
    return _hash;
  }
}
```

A.1.2 Relation

```
/**
 * @author wangz@mit.edu
 * @version 1.0
 */
import cern.colt.list.IntArrayList;
import cern.colt.map.OpenIntObjectHashMap;

public class Relation {
  // General-Purpose Relation
  protected OpenIntObjectHashMap _mapChild;
```

```
  protected OpenIntObjectHashMap _mapWeight;

  public Relation() {
    _mapChild = new OpenIntObjectHashMap();
    _mapWeight = new OpenIntObjectHashMap();
  }

  public int getParentSize() {
    return _mapChild.size();
  }

  public IntArrayList getParents() {
    return _mapChild.keys();
  }

  public int getChildSize() {
    int iSize = 0;
    IntArrayList listKey = _mapChild.keys();
    for (int i = 0; i < listKey.size(); i++) {
      IntArrayList listChild =
        (IntArrayList)_mapChild.get(listKey.getQuick(i));
      iSize += listChild.size();
    }
    return iSize;
  }

  public IntArrayList getChildren(int iParent) {
    return (IntArrayList)_mapChild.get(iParent);
  }

  public IntArrayList getWeights(int iParent) {
    return (IntArrayList)_mapWeight.get(iParent);
  }

  public void addChild(int iParent, int iChild)  {
    IntArrayList listChild = (IntArrayList)_mapChild.get(iParent);
    if (listChild == null) {
      listChild = new IntArrayList();
      _mapChild.put(iParent, listChild);
    }
    listChild.add(iChild);
  }

  public void addChild(int iParent, int iChild, int iWeight)  {
    IntArrayList listChild = (IntArrayList)_mapChild.get(iParent);
    IntArrayList listWeight = (IntArrayList)_mapWeight.get(iParent);
    if (listChild == null) {
      listChild = new IntArrayList();
      listWeight = new IntArrayList();
      _mapChild.put(iParent, listChild);
      _mapWeight.put(iParent, listWeight);
    }
    listChild.add(iChild);
    listWeight.add(iWeight);
  }
}
```

A.2 Linear Programming Solver Interfaces

```
/**
 * @author wangz@mit.edu
 * @version 1.0
 */

public class LPSolver {
  public static final int MAXIMIZE = -1;
  public static final int MINIMIZE = 1;
  public static final double LARGE_DOUBLE = 100000.0;
  public static final double SMALL_DOUBLE = 0.01;

  private NativeCPLEXAPI _native;

  public LPSolver() {
    _native = new NativeCPLEXAPI();
  }

  public void solve(int iCol, int iRow, int iSense,
                    double[] dObj, double[] dRhs, char[] cOpr,
                    int[] iMatBeg, int[] iMatCnt, int[] iMatInd,
                    double[] dMatVal,
                    double[] dLB, double[] dUB) throws Exception {
    _native.copyLPData(iCol, iRow, iSense, dObj, dRhs, cOpr,
                       iMatBeg, iMatCnt, iMatInd, dMatVal, dLB, dUB);
    _native.primOpt();
  }

  public double getObjectiveValue() throws Exception {
    return _native.getObjVal();
  }

  public double[] getDecisionValues() throws Exception {
    return _native.getX();
  }

  public double[] getDualValues() throws Exception {
    return _native.getDuals();
  }

  public void close() throws Exception {
    _native.close();
  }
}
```

APPENDIX B

B.1 Pricing Optimization Process

```
/**
 * @author wangz@mit.edu
 * @version 1.0
 */
import cern.colt.list.IntArrayList;
import cern.colt.list.DoubleArrayList;
import java.util.Hashtable;

public class PricingOptimizer {
  public PricingOptimizer() {
  }

  public void optimize(ProductDecision setX,
                       Resource setC,
                       ProductPricing setP) {
    int iSizeX = setX.getSize();
    int iSizeC = setC.getSize();
    int iSizeP = setP.getSize();
    int iRows = iSizeC + iSizeP;
    double[] dObj = new double[iSizeX];
    double[] dLB = new double[iSizeX];
    double[] dUB = new double[iSizeX];
    double[] dRhs = new double[iRows];
    char[] cOpr = new char[iRows];
    int[] iMatBeg = new int[iSizeX];
    int[] iMatCnt = new int[iSizeX];

    // Build relation between set X and set C
    Relation relationXC = new Relation();
    for (int i = 0; i < iSizeX; i++) {
      int iDuration = setX.getDuration(i);
      int iStart = setX.getDeliveryDate(i) - iDuration;
      for (int j = 0; j < iDuration; j++) {
        int iDate = iStart + j;
        String sKey = setX.getResourceID(i) + "|" + iDate;
        IntArrayList listC = setC.getRowsByKey(sKey);
        if (listC != null) { // should have only one row
          int iRow = listC.getQuick(0);
          relationXC.addChild(i, iRow);
          if (setX.isTotal(i)) {
            setC.setOrderOnHand(setX.getOrderOnHand(i)*
                                setX.getResourceUnit(i)+
                                setC.getOrderOnHand(iRow), iRow);
            setC.setDemandToCome(setX.getDemandToCome(i)*
                                 setX.getResourceUnit(i)+
                                 setC.getDemandToCome(iRow), iRow);
          }
        }
      }
    }
```

```
// Build relation between set X and set P
Relation relationXP = new Relation();
for (int i = 0; i < iSizeX; i++) {
  double dPriceX = setX.getPrice(i);
  String sKey = setX.getProductID(i)
              + "|" + setX.getDeliveryDate(i);
  IntArrayList listP = setP.getRowsByKey(sKey);
  if (listP != null) { // should have only one row
    int iRow = listP.getQuick(0);
    relationXP.addChild(i, iRow);
  }
}

// Build Objective Function and Matrix
int iSizeMat = relationXC.getChildSize()
             + relationXP.getChildSize();
System.out.println("Size of the Matrix: "+iSizeMat);
IntArrayList listInd = new IntArrayList(iSizeMat);
DoubleArrayList listVal = new DoubleArrayList(iSizeMat);
for (int i = 0; i < iSizeX; i++) {
  dObj[i] = setX.getPrice(i);
  dLB[i] = 0.0;
  dUB[i] = setX.getDemandToCome(i);
  IntArrayList listC = relationXC.getChildren(i);
  IntArrayList listP = relationXP.getChildren(i);
  iMatBeg[i] = listInd.size();
  iMatCnt[i] = listC.size() + listP.size();
  for (int j = 0; j < listC.size(); j++) {
    listInd.add(listC.getQuick(j));
    listVal.add(setX.getResourceUnit(i));
  }
  for (int j = 0; j < listP.size(); j++) {
    listInd.add(listP.getQuick(j) + iSizeC);
    listVal.add((setX.getDemandToCome(i)==0.0)?
                0.0:1.0/setX.getDemandToCome(i));
  }
}

// Build Right Hand Side
for (int i = 0; i < iSizeC; i++) {
  cOpr[i] = 'L';
  dRhs[i] = Math.max(0.0,
                     setC.getCapacity(i)-setC.getOrderOnHand(i));
}
for (int i = 0; i < iSizeP; i++) {
  cOpr[iSizeC + i] = 'L';
  dRhs[iSizeC + i] = 1.0;
}

//******** START: Use LPSolver to optimize
double dObjVal = 0.0;
try {
  LPSolver solver = new LPSolver();
  solver.solve(iSizeX, iSizeC+iSizeP, LPSolver.MAXIMIZE,
               dObj, dRhs, cOpr, iMatBeg, iMatCnt,
               listInd.elements(), listVal.elements(), dLB, dUB);
```

```
      // dObjVal = solver.getObjectiveValue();
      double[] dRes = solver.getDecisionValues();
      for (int i = 0; i < iSizeX; i++) {
        setX.setX((double)Math.round(dRes[i]*100)/100, i);
      }
      double[] dDual = solver.getDualValues();
      for (int i = 0; i < iSizeC; i++) {
        setC.setDual(dDual[i], i);
      }
      for (int i = 0; i < iSizeP; i++) {
        setP.setDual(dDual[i+iSizeC], i);
      }
      solver.close();
    } catch (Exception e) {
      e.printStackTrace();
      System.out.println(e.getMessage());
    }
    // END ********/

    for (int i = iSizeX-1; i >= 0; i--) {
      double dBidPrice = 0.0;
      IntArrayList listC = relationXC.getChildren(i);
      for (int j = 0; j < listC.size(); j++) {
        int iC = listC.getQuick(j);
        dBidPrice += setC.getDual(iC)*setX.getResourceUnit(i);
        setC.setOptimal(setX.getX(i)*setX.getResourceUnit(i)+
                        setC.getOptimal(iC), iC);
      }
      IntArrayList listP = relationXP.getChildren(i);
      int iP = listP.getQuick(0);
      if (setX.getDemandToCome(i) > 0.0)
          dBidPrice += setP.getDual(iP)/setX.getDemandToCome(i);
      dBidPrice = (double)Math.round(dBidPrice*100)/100;
      if (setX.getX(i) > 0 && setX.getPrice(i) >= dBidPrice) {
        setP.setBidPrice(dBidPrice, iP);
        setP.setPrice(setX.getPrice(i), iP);
      }
    }
  }
}
```

B.2 Pricing Optimization Test Setup

```
/**
 * @author wangz@mit.edu
 * @version 1.0
 */
import java.util.Date;

public class PricingMain {

  public static void main(String[] args) {
    int iCurrentDate = 1;
    Resource setC =
      SimulationUtil.gerenateResource(iCurrentDate);
    ProductPricing setP =
```

```
      SimulationUtil.gerenateProduct(iCurrentDate+1);
    ProductDecision setX =
      SimulationUtil.gerenateProductDemand(iCurrentDate+1);

    Date time01 = new Date();
    System.out.println(time01+" Pricing Optimization Started");

    setC.buildHash();
    setP.buildHash();
    PricingOptimizer optimizer = new PricingOptimizer();
    optimizer.optimize(setX, setC, setP);

    setX.print();
    setC.print();
    setP.print();

    Date time02 = new Date();
    System.out.println(time02+" Pricing Optimization Ended");
  }
}
```